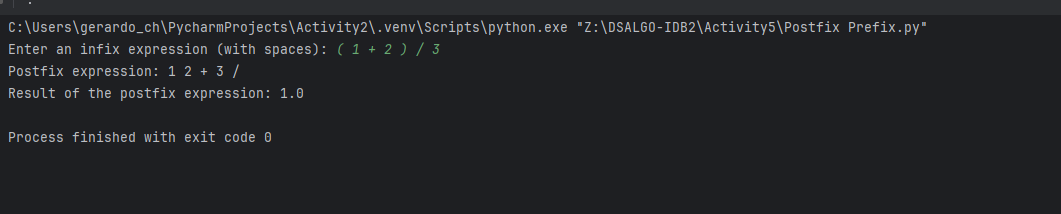
Charles Hanzel Gerardo  
DSALGO  
IDB2  
November 22, 2024  
  
Part 1:  
  
**Postfix  
notation** is an  
unambiguous way of writing an arithmetic expression without parentheses. It is  
defined so that if **“(exp1)op(exp2)”** is a normal, fully parenthesized  
expression whose operation is op, the postfix version of this is **“pexp1  
pexp2 op”,** where **pexp1** is the postfix version of **exp1** and **pexp2**  
is the postfix version of **exp2**.  
  
  
The postfix  
version of a single number or variable is just that number or variable. For  
example, the postfix version of **“((5+2) ∗ (8−3))/4” is “5 2 + 8 3 − ∗ 4 /”.** Create  
a nonrecursive way of evaluating an expression in postfix notation using python  
code.  
  
Part 2:  
  
Suppose we  
have the following numbers. **[1, 72, 81, 25, 65, 91, 11].** Write a program  
that inserts the numbers in a positional list and sorts them in **ascending  
order** first and then **descending order**using the insertion sort  
algorithm.  
  
Note: Use the LinkedStack and Positional Lists from our file tree in the Stream.

Part 1:  
  
def infix\_to\_postfix(expression):  
 precedence = {'+': 1, '-': 1, '\*': 2, '/': 2}  
 output = []  
 operators = []  
  
 tokens = expression.split()  
  
 for token in tokens:  
 if token.isdigit() or (token.replace('.', '', 1).isdigit() and token.count('.') <= 1):  
 output.append(token)  
 elif token == '(':  
 operators.append(token)  
 elif token == ')':  
 while operators and operators[-1] != '(':  
 output.append(operators.pop())  
 operators.pop()  
 elif token in precedence:  
 while (operators and operators[-1] != '(' and  
 precedence[operators[-1]] >= precedence[token]):  
 output.append(operators.pop())  
 operators.append(token)  
  
 while operators:  
 output.append(operators.pop())  
  
 return ' '.join(output)  
  
  
def evaluate\_postfix(expression):  
 stack = []  
  
 def apply\_operator(op, operand1, operand2):  
 if op == '+':  
 return operand1 + operand2  
 elif op == '-':  
 return operand1 - operand2  
 elif op == '\*':  
 return operand1 \* operand2  
 elif op == '/':  
 if operand2 == 0:  
 raise ValueError("Division by zero is not allowed")  
 return operand1 / operand2  
 else:  
 raise ValueError(f"Unsupported operator: {op}")  
  
 tokens = expression.split()  
  
 for token in tokens:  
 if token.isdigit() or (token.replace('.', '', 1).isdigit() and token.count('.') <= 1):  
 stack.append(float(token))  
 else:  
 operand2 = stack.pop()  
 operand1 = stack.pop()  
 result = apply\_operator(token, operand1, operand2)  
 stack.append(result)  
  
 return stack.pop()  
  
  
def main():  
 infix\_expression = input("Enter an infix expression (with spaces): ")  
  
 try:  
 postfix\_expression = infix\_to\_postfix(infix\_expression)  
 print(f"Postfix expression: {postfix\_expression}")  
  
 result = evaluate\_postfix(postfix\_expression)  
 print(f"Result of the postfix expression: {result}")  
 except Exception as e:  
 print(f"Error: {e}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()



Part 2:  
  
class PositionalList:  
  
 class Position:  
 def \_\_init\_\_(self, element, prev=None, next=None):  
 self.element = element  
 self.prev = prev  
 self.next = next  
  
 def \_\_init\_\_(self):  
 self.head = None  
 self.tail = None  
 self.size = 0  
  
 def is\_empty(self):  
 return self.size == 0  
  
 def first(self):  
 if self.is\_empty():  
 raise IndexError("The list is empty.")  
 return self.head  
  
 def last(self):  
 if self.is\_empty():  
 raise IndexError("The list is empty.")  
 return self.tail  
  
 def add\_last(self, element):  
 new\_node = self.Position(element, self.tail, None)  
 if self.is\_empty():  
 self.head = self.tail = new\_node  
 else:  
 self.tail.next = new\_node  
 self.tail = new\_node  
 self.size += 1  
  
 def remove(self, pos):  
 if pos is self.head:  
 self.head = self.head.next  
 if pos is self.tail:  
 self.tail = self.tail.prev  
 if pos.prev:  
 pos.prev.next = pos.next  
 if pos.next:  
 pos.next.prev = pos.prev  
 self.size -= 1  
  
 def insert\_after(self, pos, element):  
 new\_node = self.Position(element, pos, pos.next)  
 if pos.next:  
 pos.next.prev = new\_node  
 pos.next = new\_node  
 if pos == self.tail:  
 self.tail = new\_node  
 self.size += 1  
  
 def \_\_iter\_\_(self):  
 current = self.head  
 while current:  
 yield current.element  
 current = current.next  
  
  
class LinkedStack:  
  
 class Node:  
 def \_\_init\_\_(self, element, next=None):  
 self.element = element  
 self.next = next  
  
 def \_\_init\_\_(self):  
 self.\_top = None  
 self.\_size = 0  
  
 def push(self, element):  
 new\_node = self.Node(element, self.\_top)  
 self.\_top = new\_node  
 self.\_size += 1  
  
 def pop(self):  
 if self.is\_empty():  
 raise IndexError("pop from empty stack")  
 popped\_element = self.\_top.element  
 self.\_top = self.\_top.next  
 self.\_size -= 1  
 return popped\_element  
  
 def peek(self):  
 if self.is\_empty():  
 raise IndexError("peek from empty stack")  
 return self.\_top.element  
  
 def is\_empty(self):  
 return self.\_size == 0  
  
 def size(self):  
 return self.\_size  
  
  
def insertion\_sort(lst, ascending=True):  
 current = lst.head  
 while current is not None:  
 key = current  
 current = current.next  
 while key.prev and (key.prev.element > key.element if ascending else key.prev.element < key.element):  
 key.element, key.prev.element = key.prev.element, key.element  
 key = key.prev  
  
  
def main():  
 user\_input = input("Enter numbers separated by spaces: ")  
 numbers = list(map(int, user\_input.split()))  
  
 pos\_list = PositionalList()  
 for num in numbers:  
 pos\_list.add\_last(num)  
  
 print("Original list:", list(pos\_list))  
 insertion\_sort(pos\_list, ascending=True)  
 print("Sorted in Ascending Order:", list(pos\_list))  
  
 insertion\_sort(pos\_list, ascending=False)  
 print("Sorted in Descending Order:", list(pos\_list))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

